Timers

Profs. Karkal Prabhu and Naga Kandasamy
ECE Department, Drexel University

This lab assignment consists of three problems dealing with timers. Complete the first problem in the lab and show its output to the teaching assistant to be checked for correctness. You may complete the remaining two problems outside of lab and submit via BBLearn. You will find detailed submission instructions towards the end of this document. Your C code must be clearly written, properly formatted, and well commented for full credit.

Brief Primer on Timers

Before discussing the programming assignments, let's review some important concepts related to timers.[1] We focus our discussion on the 16-bit `Timer_A` modules in the MSP432 microprocessor. There are four such identical timer modules named `TA0`, `TA1`, `TA2`, and `TA3`. Each of these timers can be fed via four different clocks.

Choosing an appropriate clock signal to supply to the timer is a fundamental step. Therefore, let's understand the clock system on the MSP432. Figure 1 shows the high-level block diagram. There are seven independent oscillator sources which can be provided as inputs to the clock system:

- *External low-frequency crystal oscillator clock,* LFXT CLK, that can supply a signal up to 32KHz.
- *External high-frequency crystal oscillator clock,* HFXT CLK, that can be configured to supply a signal between 1MHz to 48MHz.
- *Internal digitally controlled clock,* DCO, that can be configured to supply a signal between 1MHz to 48MHz. This serves as the default clock for the processor.
- *Internal low-frequency oscillator clock,* VLO CLK, that typically supplies a 9.4KHz signal.
- *Internal low-frequency oscillator clock,* REFO CLK, that can be configured to supply a 32KHz to 128KHz signal.
- *Internal low-power oscillator,* MOD OSC, that supplies a 25MHz signal.
- *Internal low-frequency oscillator,* SYS OSC, that supplies a 5MHz signal.

The clock module synthesizes the above-described oscillator signals as per Fig. 1 and supplies various output signals that are used by the main microprocessor as well as by the various peripheral modules such as timers, pulse-width modulators, UART, and analog/digital converters. These output clock signals include:

[1] Please read through Chapter 7 of the textbook for a more detailed description of timing operations on the MSP432.

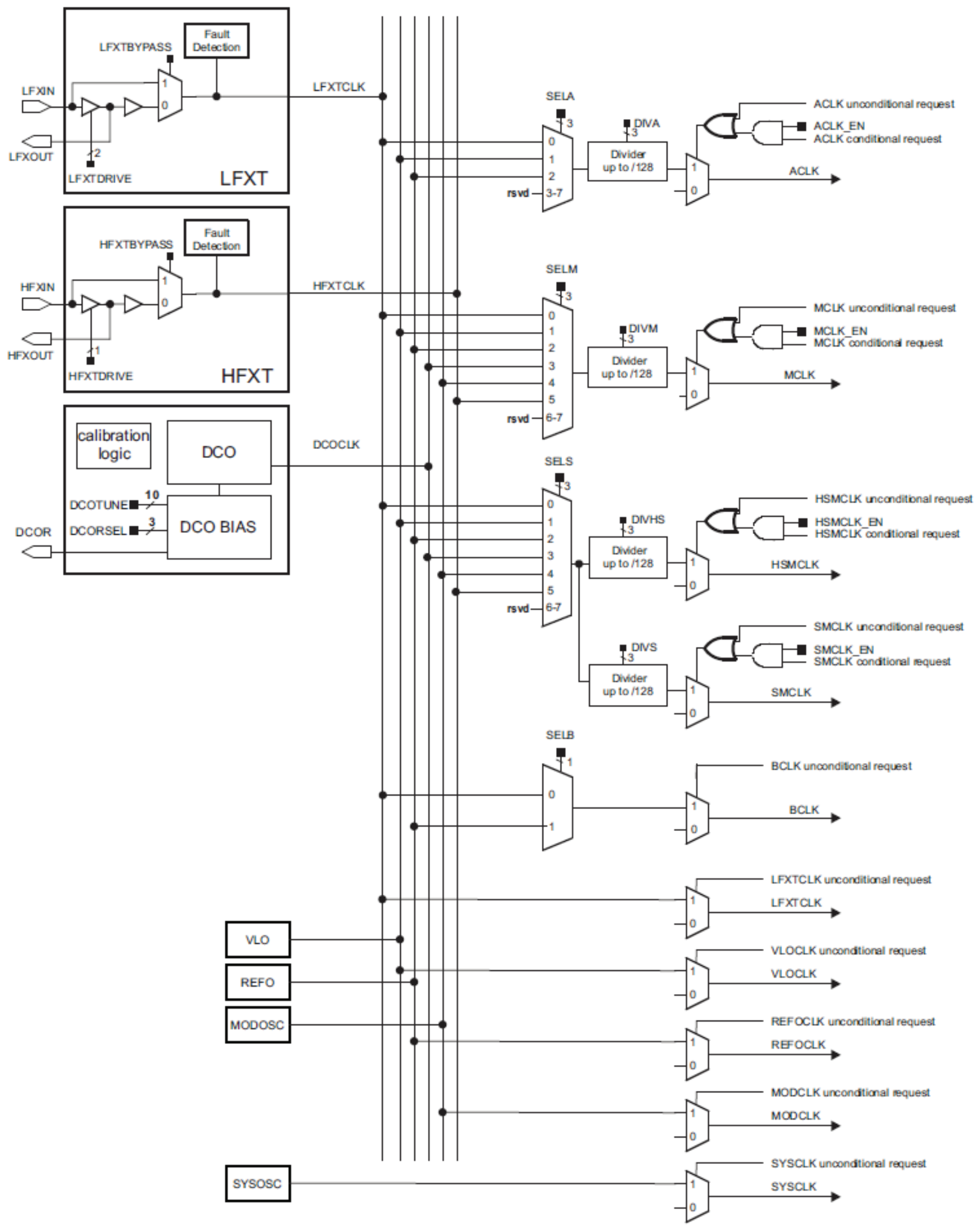

Fig. 1: Block diagram of the clock system on the MSP432, taken from Chapter 5 of the MSP432 Technical Reference Manual.

- *Master clock,* MCLK, that can be selected from one of the following clock sources via software: LFXT CLK, VLO CLK, REFO CLK, DCO CLK, MOD CLK, or HFXT CLK. The selected source can be further divided by a factor of 1, 2, 4, 8, 16, 32, or 64. This clock is used by the microprocessor and can also be directly used by some peripheral modules such as the UART.

- *Subsystem master clock,* HSMCLK, that can be selected from one of the following clock sources via software: LFXT CLK, VLO CLK, REFO CLK, DCO CLK, MOD CLK, or

HFXT CLK. The selected source can be further divided by a factor of 1, 2, 4, 8, 16, 32, or 64. This clock can be used by individual peripheral modules.

- *Low-speed subsystem master clock,* SMCLK, that can be selected from one of the following clock sources via software: LFXT CLK, VLO CLK, REFO CLK, DCO CLK, MOD CLK, or HFXT CLK. The selected source can be further divided by a factor of 1, 2, 4, 8, 16, 32, or 64. SMCLK is limited to a maximum of 24MHz and can be selected as the clock source for individual peripheral modules.
- *Auxiliary clock,* ACLK, that can be selected from one of the following clock sources via software: LFXTCLK, VLOCLK, or REFOCLK. The selected source can be further divided by a factor of 1, 2, 4, 8, 16, 32, or 64. This signal is selectable by individual peripheral modules and is limited to a maximum of 128 kHz.
- *Low-speed backup domain clock,* BCLK that can be selected from one of the following clock sources via software: LFXTCLK and REFOCLK. This clock is primarily used in the backup domain and is limited to a maximum of 32.768 kHz.[2]

In our example, we will run our timers off SMCLK and ACLK, where these clocks themselves are derived from the DCO and REFO oscillators, respectively. The following code snippet accomplishes this. The DCO and REFO oscillators are set to 12MHz and 32KHz respectively.

```
1 CS_setDCOCenteredFrequency(CS_DCO_FREQUENCY_12); /* DCO = 12MHz. */
2 /* MCLK, HSMCLK, and SMCLK derive from DCO. */
3 CS_initClockSignal(CS_MCLK, CS_DCOCLK_SELECT, CS_CLOCK_DIVIDER_1);
4 CS_initClockSignal(CS_HSMCLK, CS_DCOCLK_SELECT, CS_CLOCK_DIVIDER_1);
5 CS_initClockSignal(CS_SMCLK, CS_DCOCLK_SELECT, CS_CLOCK_DIVIDER_1);
6 /* Derive the low-speed auxiliary clock, ACLK, from reference
7  oscillator set to 32KHz. */
8 CS_setReferenceOscillatorFrequency(CS_REFO_32KHZ);
9 CS_initClockSignal(CS_ACLK, CS_REFOCLK_SELECT, CS_CLOCK_DIVIDER_1);
```

Note that we also require our program to display information on the terminal via the UART which is also clocked by SMCLK. Recall that to operate the UART at 9600 bps, a 12MHz clock must be supplied to it. So, if we choose a different DCO frequency in line 1, for example, 48MHz, then we must increase the clock-divider value for SMCLK by a factor of four, that is, set the second parameter to `CS_CLOCK_DIVIDER_4` in line 5.

The next step is to configure the timers. We will see how to operate timers in three different modes of operation.

Continuous mode. Let's configure `Timer_A0` to operate in continuous mode, which is shown in Fig. 2 in which the 16-bit timer counts up until it reaches its maximum value of 0xFFFF or 65535 in decimal, then restarts again from zero. The Timer_A Interrupt FlaG or TAIFG bit is set when the counter value changes from 0xFFFF to zero, which will trigger `TA0_N_IRQHandler()` if this

[2]The MSP432 allows for program state to be backed-up to Flash memory, for example when the battery source powering the microprocessor is low.

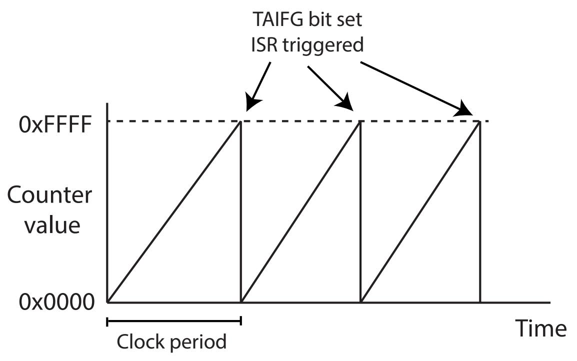

Fig. 2: Operation of the timer module in continuous mode. The TAIFG bit is set when the counter overflows, from 0xFFFF to zero.

ISR is enabled in the program. The timer period when operating in this mode is

$$p = \frac{2^{16}}{f_{\mathrm{CLK}}} = \frac{65536}{f_{\mathrm{CLK}}},$$

where f_{CLK} is the frequency of the clock signal supplied to the timer. The following snippet of code operates the timer in continuous mode.

```c
const Timer_A_ContinuousModeConfig continuousModeConfig = {
    TIMER_A_CLOCKSOURCE_SMCLK,          // Clock source, f_CLK
    TIMER_A_CLOCKSOURCE_DIVIDER_64,     // Clock divider
    TIMER_A_TAIE_INTERRUPT_ENABLE,      // TAIE interrupt enabled
    TIMER_A_DO_CLEAR
};

int overflow = 0;
int main(void)
{
    /* Configure the continuous-mode timer. */
    Timer_A_configureContinuousMode(TIMER_A0_BASE,
                    &continuousModeConfig);
    /* Enable the interrupt processing system. */
    Interrupt_enableInterrupt(INT_TA0_N);
    Interrupt_enableMaster();
    /* Start the counter. */
    Timer_A_startCounter(TIMER_A0_BASE, TIMER_A_CONTINUOUS_MODE);

    /* Other code. */
}

void TA0_N_IRQHandler(void) /* ISR handling counter overflow. */
{
    Timer_A_clearInterruptFlag(TIMER_A0_BASE);
    overflow++;
}
```

The timer is clocked using SMCLK with a 64:1 clock divider, in this example. Since SMCLK is derived from the DCO that is operating at 12MHz with a 1:1 ratio, our timer's period as per this configuration is

$$p = \frac{2^{16}}{12000000 \times \frac{1}{64}} = 0.35s.$$

We also enable the ISR which is executed each time the counter value overflows from 0xFFFF to 0X0000, generating the TAIE interrupt. The ISR code resets the interrupt flag and keeps track of the number of times the counter overflows during the course of program execution.

Let's now discuss how to measure elapsed time T using our timer. Suppose we wish to measure the execution time of a code block. We can read the counter values as follows.

```
/* Start the timer. */
Timer_A_startCounter(TIMER_A0_BASE, TIMER_A_CONTINUOUS_MODE);

uint16_t t1 = Timer_A_getCounterValue(TIMER_A0_BASE);

/* Code block to be timed. */

uint16_t t2 = Timer_A_getCounterValue(TIMER_A0_BASE);
```

In general, the elapsed time cannot be simply calculated as

$$T = \frac{t_2 - t_1}{f_{\text{CLK}}}$$

since the counter might have overflowed multiple times during execution of the code block. Therefore, we must take into account the number of overflows as well when calculating the elapsed time. Let us denote this quantity as n, where $n \geq 0$. The elapsed-time equation then becomes

$$T = \frac{(2^{16} - t_1) + (t_2 - 0) + (n - 1) \times 2^{16}}{f_{\text{CLK}}}.$$

Let's examine the numerator term more closely. We obtain the starting counter value as t_1. So, if the counter had indeed overflowed during program execution, then prior to the first overflow, that is during the first timer period we would have $2^{16} - t_1$ counts. The ending counter value is obtained as t_2 and therefore during the last timer period, we would have $t_2 - 0$ counts. Between the first and last periods, we account for $n - 1$ timer overflows.

As a sanity check of the above equation, when the counter does not overflow and both t_1 and t_2 are obtained within the very first timer period, we can set $n = 0$ to obtain elapsed time as

$$T = \frac{(2^{16} - t_1) + (t_2 - 0) - 1 \times 2^{16}}{f_{\text{CLK}}} = \frac{t_2 - t_1}{f_{\text{CLK}}},$$

which is indeed the value we expect.

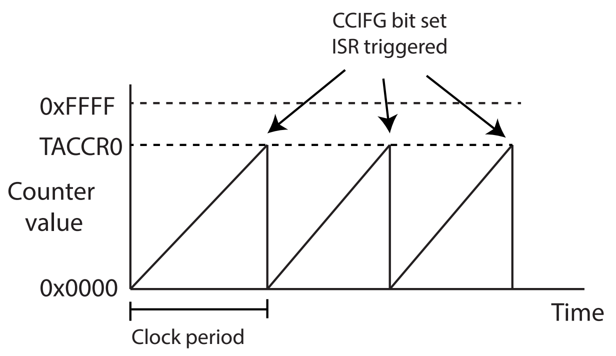

Fig. 3: Operation of the timer module in up mode. The CCIFG bit is set when the counter reaches the compare value in `TACCR0` and the TAIFG bit is set when the counter resets to zero.

The precision of the time measurement can be controlled via the input clock frequency f_{CLK} that is supplied to the timer.

Up mode. Let's configure `Timer_A1` to operate in up mode. As shown in Fig. 3, in this mode the timer counts up until it reaches the value stored in the *compare* register, `TACCR0`, located within the timer module. Then it restarts from zero again. The TAIFG bit is set in the internal status register when the counter value changes from `TACCR0` to zero; and so is the Capture/Compare Interrupt FlaG or CCIFG when the counter value changes from `TACCR0 - 1` to `TACCR0`.

The following ISRs can be associated with `Timer_A1` in up mode:

- The `TA1_0_IRQHandler()` ISR is invoked whenever the CCIFG bit for `TACCR0` is set, that is when the counter value reaches the compare value.
- The `TA0_N_IRQHandler()` ISR is invoked whenever the TAIFG bit is set, which occurs when the counter value changes to zero. This bit can also be set by other compare registers within the timer module, except for `TACCR0`.

The timer period when operating in the up mode is a function of the `TACCR0` value,

$$p = \frac{\texttt{TACCR0} + 1}{f_{\text{CLK}}},$$

where f_{CLK} is the frequency of the clock signal supplied to the timer. The following snippet of code operates the timer in up mode.

```
#define TIMER_A1_COUNTER 31999
const Timer_A_UpModeConfig upConfig=
{
    TIMER_A_CLOCKSOURCE_ACLK,              // Clock source, f_CLK
    TIMER_A_CLOCKSOURCE_DIVIDER_1,         // Clock divider
    TIMER_A1_COUNTER,                      // Counter value
    TIMER_A_TAIE_INTERRUPT_DISABLE,        // TAIE interrupt disabled
    TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE,    // CCIE interrupt enabled
    TIMER_A_DO_CLEAR
};
```

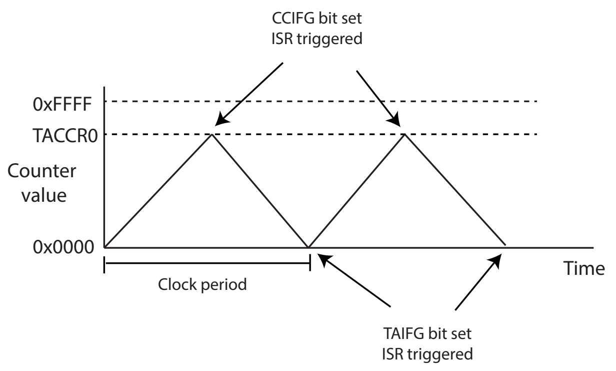

Fig. 4: Operation of the timer module in up/down mode. The CCIFG bit is set when the counter reaches the compare value in `TACCR0` and the TAIFG bit is set when the counter reaches zero.

```
int overflow = 0;

int main(void)
{
    /* Configure the up-mode timer. */
    Timer_A_configureUpMode(TIMER_A1_BASE, &upConfig);
    /* Enable the interrupt processing system to catch CCIFG. */
    Interrupt_enableInterrupt(INT_TA1_0);
    Interrupt_enableMaster();
    /* Start the counter. */
    Timer_A_startCounter(TIMER_A1_BASE, TIMER_A_UP_MODE);

    /* Other code. */
}

void TA1_0_IRQHandler(void)  /* ISR handling CCIFG flag. */
{
    Timer_A_clearCaptureCompareInterrupt(TIMER_A1_BASE,
                TIMER_A_CAPTURECOMPARE_REGISTER_0);
    overflow++;
    return;
}
```

`Timer_A1` is clocked 1:1 using ACLK which is derived from the low-frequency reference oscillator operating at 32KHz. The ISR clears the pending capture/compare interrupt and keeps track of the number of times the counter overflows, with respect to the compare value. The period of the timer is one second.

Up/Down mode. As shown in Fig. 4, the timer first counts up until it reaches the value stored in `TACCR0`, upon which the counting is inverted and the timer counts down from `TACCR0` to zero. This process is repeated.

The TAIFG bit is set when the counter value changes from one to zero when counting down and the CCIFG bit is set when the counter value changes from `TACCR0 - 1` to `TACCR0` when counting up. The following ISRs can be associated with a timer, say `Timer_A2`, configured in up/down mode:

- The `TA2_0_IRQHandler()` ISR is invoked whenever the CCIFG bit for `TACCR0` is set, when the counter value reaches the compare value when counting up.
- The `TA2_N_IRQHandler()` ISR is invoked whenever the TAIFG bit is set, which occurs when the counter value becomes zero when counting down. This bit can also be set by other compare registers within the timer module, except for `TACCR0`.

The timer period when operating in the up/down mode is

$$p = \frac{2 \times \texttt{TACCR0}}{f_{\mathrm{CLK}}},$$

where f_{CLK} is the frequency of the clock signal supplied to the timer. The following snippet of code operates `TImer_A2` in up/down mode.

```
#define TIMER_A2_COUNTER 16000

const Timer_A_UpDownModeConfig updownConfig = {
    TIMER_A_CLOCKSOURCE_ACLK,           // Clock source, f_CLK
    TIMER_A_CLOCKSOURCE_DIVIDER_1,      // Clock divider = 1
    TIMER_A2_COUNTER,                   // Compare value
    TIMER_A_TAIE_INTERRUPT_ENABLE,      // TAIE interrupt enabled
    TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE,// CCIE interrupt enabled
    TIMER_A_DO_CLEAR
};

int overflow = 0;

int main(void)
{
    /* Configure the up/down-mode timer. */
    Timer_A_configureUpDownMode(TIMER_A2_BASE, &upDownConfig);
    /* Enable interrupt processing system to catch TAIFG
       and CCIFG. */
    Interrupt_enableInterrupt(INT_TA2_0);
    Interrupt_enableInterrupt(INT_TA1_N);
    Interrupt_enableMaster();
    /* Start the counter. */
    Timer_A_startCounter(TIMER_A2_BASE, TIMER_A_UP_DOWN_MODE);

    /* Other code. */
}



void TA2_0_IRQHandler(void) /* ISR handling CCIFG flag. */
```

```
{
    Timer_A_clearCaptureCompareInterrupt(TIMER_A2_BASE,
                TIMER_A_CAPTURECOMPARE_REGISTER_0);
    overflow++;
    return;
}

void TA2_N_IRQHandler(void) /* ISR handling TAIFG flag. */
{
    Timer_A_clearInterruptFlag(TIMER_A2_BASE);
    /* ISR specific code. */
    return;
}
```

`Timer_A2` is clocked 1:1 using ACLK which is derived from the low-frequency reference oscillator operating at 32KHz. Two ISRs are provided to respond to the CCIFG and TAIFG flags. The period of the timer is one second.

Halt mode. In this mode, the timer stops counting, but retains its current value to continue later when restarted. The `Timer_A` modules are initially placed in this state to save power. The program has to explicitly start the counter.

The above-discussed code snippets have been consolidated in a CCS project for the MSP432 called `timer_operation` that is available via BBLearn. The program configures three timers modules, `Timer_A0`, `Timer_A1`, and `Timer_A2` in continuous, up, and up/down modes, respectively. `Timer_A0` toggles the red LED1 at the set period, whereas `Timer_A1` and `Timer_A2` toggle the red and green components of the RGB LED2. You may use this code as the starting point for developing your programming assignments.

Measuring Elapsed Time

(10 points) Using the CCS skeleton project called `elapsed_time`, available via BBLearn, develop an MSP432 program that measures the elapsed time, in seconds, between two consecutive presses of buttons S1 and S2. Your program must configure and start a timer in continuous mode. It must then wait for the user to press S1, upon which the current counter value is read. The program then waits for S2 to be pressed, upon which the counter value is read again. When calculating the elapsed time between these button presses, your program must take counter overflows into account. Print out the elapsed time on the terminal, rounding the value to the nearest second.

Signature of the teaching assistant: ____________ Date: ____________

Timing Function Execution Time

(10 points) You will develop timing functions that use the timer module to measure the execution time of a compute-intensive function. Consider the following function that multiplies two `dim` × `dim` matrices whose members are integer values:

```
void matrixMult(int *A, int *B, int *C, unsigned int dim)
{
    unsigned int i, j, k;
    int temp;
    for (i = 0; i < dim; i++) {
        for (j = 0; j < dim; j++) {
            temp = 0.0;
            for (k = 0; k < dim; k++)
                temp += A[i * dim + k] * B[k * dim + j];

            C[i * dim + j] = temp;
        }
    }

    return;
}
```

You are asked to write two functions, `tic()` and `toc()`, that work in concert to measure the execution time incurred by `matrixMult` in milliseconds. The skeleton of the program provided to you is as follows.

```
/* Matrix dimensions and the min and max values in the matrix. */
#define MATRIX_SIZE 40
#define MIN_VALUE 5
#define MAX_VALUE 10

/* Global variables: arrays for the matrices. */
int A[MATRIX_SIZE * MATRIX_SIZE];
int B[MATRIX_SIZE * MATRIX_SIZE];
int C[MATRIX_SIZE * MATRIX_SIZE];

void tic(void) /* The tic function. */
{
    /* FIXME: Complete the function. */
    return;
}

void toc(void) /* The toc function. */
{
    /* FIXME: Complete the function. */
    return;
}
```

```c
/* Function implements delay for specified number of millseconds. */
void delay(unsigned int msecs)
{
    unsigned int i;
    for (i = 0; i < 275 * msecs; i++);
    return;
}

int main(void)
{
    /* Stop Watchdog  */
    MAP_WDT_A_holdTimer();

    /* Initialize the clock system. */
    CS_setDCOCenteredFrequency(CS_DCO_FREQUENCY_12); // DCO =  12 MHz
    CS_initClockSignal(CS_MCLK, CS_DCOCLK_SELECT, CS_CLOCK_DIVIDER_1);
    CS_initClockSignal(CS_HSMCLK, CS_DCOCLK_SELECT,
                       CS_CLOCK_DIVIDER_1);
    CS_initClockSignal(CS_SMCLK, CS_DCOCLK_SELECT,
                       CS_CLOCK_DIVIDER_1);

    /* Initialize UART module. */
    initUART();
    writeString("Established communication with the board");

    srand(rand()); /* Seed random number generator. */

    while (1) {
        /* Populate matrices with random numbers. */
        populateMatrix(A, MIN_VALUE, MAX_VALUE, MATRIX_SIZE);
        populateMatrix(B, MIN_VALUE, MAX_VALUE, MATRIX_SIZE);

        writeString("\n\rMultiplying matrices");

        tic(); /* Time the multiplication operation. */
        matrixMult(A, B, C, MATRIX_SIZE);
        toc();

        writeString("\n\rDone multiplying");

        /* FIXME: Display elapsed time on the terminal. */

        delay(1000); /* Delay 1s before starting again. */
    }
}
```

In the above code, a random number generator is initialized via the `srand` function and the matrices are populated with random integer values during each multiplication run as follows.

```c
void populateMatrix(int *M, int min, int max, unsigned int dim)
{
    unsigned int i, j;
    for (i = 0; i < dim; i++) {
        for (j = 0; j < dim; j++) {
            M[i * dim + j] = ceil (min + (max - min) *
                                    rand ()/(float) RAND_MAX);
        }
    }

    return;
}
```

Using the skeleton CCS project called `timing_functions`, available from BBLearn, as the starting point, complete the program to achieve the desired functionality. The DCO is set to 12MHz in the code. Do not change this setting. You may however add additional functions to your code as you see fit. The `tic()` and `toc()` functions must measure the elapsed time after each run of `matrixMult` to within one millisecond accuracy, and the result must be stored in the variable `elapsedTime` after the function `toc` returns and subsequently displayed on the terminal. Display the elapsed times when multiplying matrices of the following sizes: 10×10, 25×25, and 40×40.

Implementation of a Combination Lock

(40 points) Consider a combination lock system with a keypad having five push buttons labeled ‘1’, ‘2’, ‘3’, ‘4’, and ‘5.’ For the purposes of this problem, your computer keyboard will serve as the keypad. We start with the system in an UNLOCKED state. In this state, the user can press ‘1’ to lock the system, that is put it in a LOCKED state. In this state, the red LED on the MSP432 board must light up—light up solidly, not blink—and a prompt must be displayed on the terminal requesting that the user enter a five-digit combination code to unlock the system:

```
Please enter code to unlock the system:
```

To unlock the system, we must read the user input, one button push at a time from the keyboard, and check for validity. If the entered code is valid, the LED is switched off (simulating the system being unlocked). For example, if the valid sequence is ‘2’, ‘4’, ‘1’, ‘5’, ‘3’, then one can develop a finite state machine to check if the user input matches this combination. The following function develops such a state machine; it returns 1 if a valid combination is entered, 0 otherwise.

```
int getCombination()
{
   int currentState = -1;  /* Our start state. */
   char input;
   while (1) {
      input = getButton();  /* Get button pushed by user via UART. */
      switch (input) {
            case `1':
                if (currentState == 4)
                    currentState = 1;
                else
                    return 0;
                break;

            case `2':
                if (currentState == -1)
                    currentState = 2;
                else
                    return 0;
                break;

            case `3':
                if (currentState == 5)
                    return 1;
                else
                    return 0;
                break;

            case `4':
                if (currentState == 2)
                    currentState = 4;
                else
                    return 0;
                break;

            case `5':
                if (currentState == 1)
                    currentState = 5;
                else
                    return 0;
                break;

            default:
                return 0;
         }
     }
}
```

Starting with the above code snippet, develop the following enhancements as part of your program. The combination entered by the user must be processed one button-push at a time.

- **(5 points)** If in the UNLOCKED state, the user must be able to lock the system by pushing '1' anytime.
- **(5 points + 10 bonus points)** Limit the number of unsuccessful attempts allowed by locking out the user permanently after three consecutive failed attempts. In this state, the system must not respond any more to user input. (For extra credit, implement a SUPERVISOR state in which the system can only be unlocked by a supervisor using a different five-digit code.)
- **(10 points)** Impose a time limit on entering the correct combination in that after the user enters the first symbol of the combination, he or she has five seconds to enter the entire sequence. If the time expires during this process, the user has to enter the entire code again from the beginning.
- **(10 points)** Finally, the finite-state machine developed in the function assumes that the combination to the lock is hard-coded. Relax this assumption by extending the functionality of the program to accept new combinations at run time as follows. In the UNLOCKED state, the user must be able to push the button '2' to reprogram a new five-digit combination for the system. The system displays a

  ```
  Enter New Key:
  ```

  prompt on the terminal. After a new five-digit code is entered, the screen displays

  ```
  Enter Key Again to Confirm:
  ```

  and the user enters the code again. If the code is confirmed, system goes back to the UNLOCKED state and the new key is used from hereon. Otherwise the system displays an error and remains in the UNLOCKED state while maintaining the old key. During the reprogramming phase, the LED blinks.

Submission Instructions

Once you have implemented all of the required features described for Problems 2 and 3, submit your code by doing the following:

- Run `Clean Project` to remove the executable and object files from the project folders. We must be able to build your projects from source and we don't require your pre-compiled executables or intermediate object files. **If your code does not at the very least compile, you will receive a zero.**
- Zip up each project separately and upload both zip files using the Blackboard Learn submission link found on the course website.